CHAPTER 1

Getting Familiar with Python

Python is an open source programming language created by a Dutch programmer named Guido van Rossum. Named after the British comedy group Monty Python, Python is a high-level, interpreted, open source language and is one of the most sought-after and rapidly growing programming languages in the world today. It is also the language of preference for data science and machine learning.

In this chapter, we first introduce the Jupyter notebook – a web application for running code in Python. We then cover the basic concepts in Python, including data types, operators, containers, functions, classes and file handling and exception handling, and standards for writing code and modules.

The code examples for this book have been written using Python version 3.7.3 and Anaconda version 4.7.10.

Technical requirements

Anaconda is an open source platform used widely by Python programmers and data scientists. Installing this platform installs Python, the Jupyter notebook application, and hundreds of libraries. The following are the steps you need to follow for installing the Anaconda distribution.

1. Open the following URL: `https://www.anaconda.com/products/individual`
2. Click the installer for your operating system, as shown in Figure 1-1. The installer gets downloaded to your system.

G. Rajagopalan, *A Python Data Analyst's Toolkit*, https://doi.org/10.1007/978-1-4842-6399-0_1

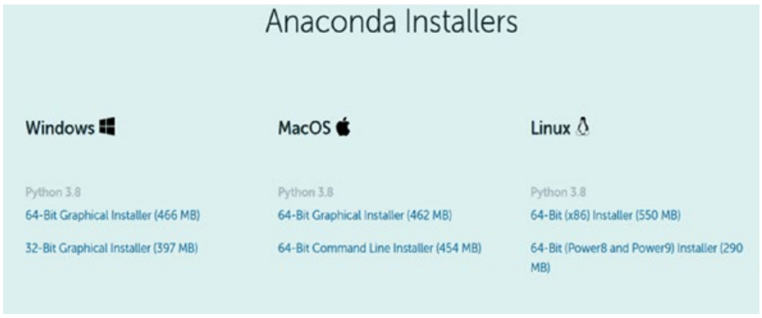

Anaconda Installers
Windows
MacOS
Linux
Python 3.8
64-Bit Graphical Installer (466 MB)
32-Bit Graphical Installer (397 MB)
Python 3.8
64-Bit Graphical Installer (462 MB)
64-Bit Command Line Installer (454 MB)
Python 3.8
64-Bit (x86) Installer (550 MB)
64-Bit (Power8 and Power9) Installer (290 MB)

Figure 1-1. *Installing Anaconda*

3. Open the installer (file downloaded in the previous step) and run it.
4. After the installation is complete, open the Jupyter application by typing “jupyter notebook” or “jupyter” in the explorer (search bar) next to the start menu, as shown in Figure 1-2 (shown for Windows OS).

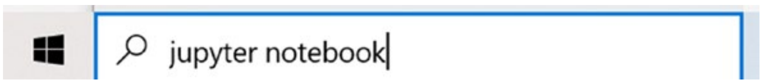


Figure 1-2. *Launching Jupyter*

Please follow the following steps for downloading all the data files used in this book:

- Click the following link: `https://github.com/DataRepo2019/Data-files`
- Select the green “Code” menu and click on “Download ZIP” from the dropdown list of this menu
- Extract the files from the downloaded zip folder and import these files into your Jupyter application

Now that we have installed and launched Jupyter, let us understand how to use this application in the next section.

Getting started with Jupyter notebooks

Before we discuss the essentials of Jupyter notebooks, let us discuss what an integrated development environment (or IDE) is. An IDE brings together the various activities involved in programming, like including writing and editing code, debugging, and creating executables. It also includes features like autocompletion (completing what the user wants to type, thus enabling the user to focus on logic and problem-solving) and syntax highlighting (highlighting the various elements and keywords of the language). There are many IDEs for Python, apart from Jupyter, including Enthought Canopy, Spyder, PyCharm, and Rodeo. There are several reasons for Jupyter becoming a ubiquitous, de facto standard in the data science community. These include ease of use and customization, support for several programming languages, platform independence, facilitation of access to remote data, and the benefit of combining output, code, and multimedia under one roof.

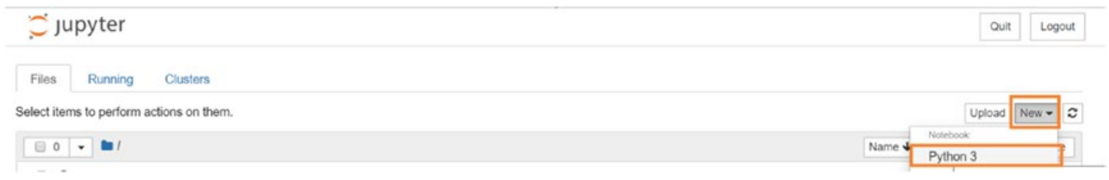
JupyterLab is the IDE for Jupyter notebooks. Jupyter notebooks are web applications that run locally on a user's machine. They can be used for loading, cleaning, analyzing, and modeling data. You can add code, equations, images, and markdown text in a Jupyter notebook. Jupyter notebooks serve the dual purpose of running your code as well as serving as a platform for presenting and sharing your work with others. Let us look at the various features of this application.

1. **Opening the dashboard**

   Type "jupyter notebook" in the search bar next to the start menu. This will open the Jupyter dashboard. The dashboard can be used to create new notebooks or open an existing one.

2. **Creating a new notebook**

   Create a new Jupyter notebook by selecting *New* from the upper right corner of the Jupyter dashboard and then select *Python 3* from the drop-down list that appears, as shown in Figure 1-3.

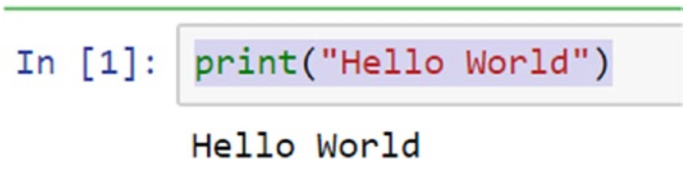


Select items to perform actions on them.

Figure 1-3. *Creating a new Jupyter notebook*

3. **Entering and executing code**

   Click inside the first cell in your notebook and type a simple line of code, as shown in Figure 1-4. Execute the code by selecting *Run Cells* from the "Cell" menu, or use the shortcut keys *Ctrl+Enter*.



```
In [1]: print("Hello World")

        Hello World
```

Figure 1-4. *Simple code statement in a Jupyter cell*

4. **Adding markdown text or headings**

   In the new cell, change the formatting by selecting *Markdown* as shown in Figure 1-5, or by pressing the keys *Esc+M* on your keyboard. You can also add a heading to your Jupyter notebook by selecting *Heading* from the drop-down list shown in the following or pressing the shortcut keys *Esc+(1/2/3/4)*.

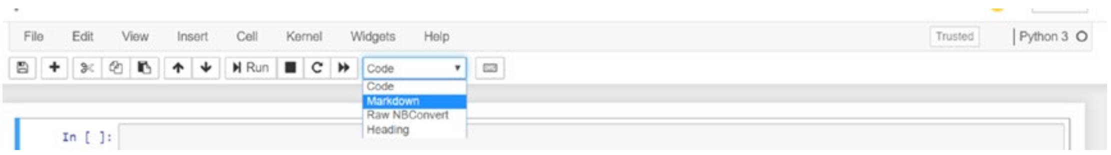


Figure 1-5. *Changing the mode to Markdown*

5. **Renaming a notebook**

   Click the default name of the notebook and type a new name, as shown in Figure 1-6.

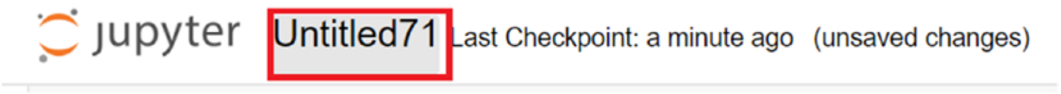


Figure 1-6. *Changing the name of a file*

   You can also rename a notebook by selecting *File* ➤ *Rename*.

6. **Saving a notebook**

   Press Ctrl+S or choose File ➤ Save and Checkpoint.

7. **Downloading the notebook**

   You can email or share your notebook by downloading your notebook using the option *File* ➤ *Download as* ➤ *notebook (.ipynb)*, as shown in Figure 1-7.

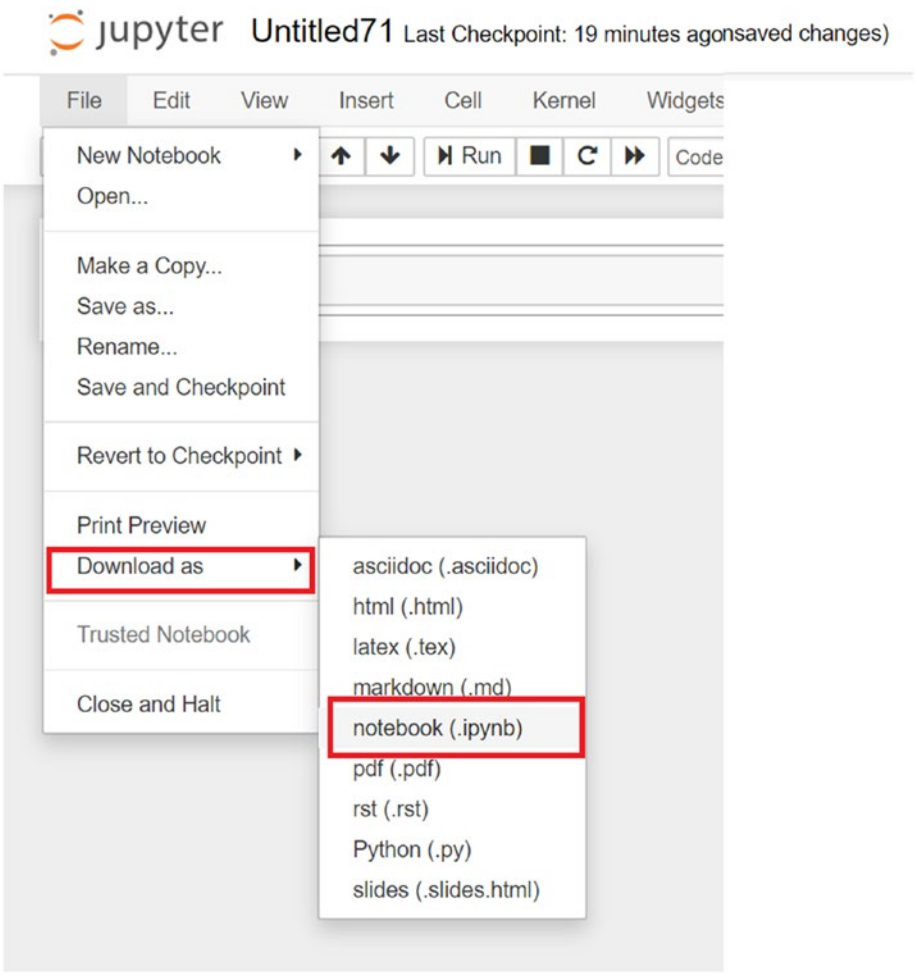


Figure 1-7. *Downloading a Jupyter notebook*

Shortcuts and other features in Jupyter

Let us look at some key features of Jupyter notebooks, including shortcuts, tab completions, and magic commands.

Table 1-1 gives some of the familiar icons found in Jupyter notebooks, the corresponding menu functions, and the keyboard shortcuts.

Table 1-1. Jupyter Notebook Toolbar Functions

Icon in Toolbar	**Function**	**Keyboard shortcut**	**Menu function**
💾	Saving a Jupyter notebook	*Esc+s*	*File ➤ Save as*
➕	Adding a new cell to a Jupyter notebook	*Esc+b* (adding a cell below the current cell), or *Esc+a* (adding a cell above the current cell)	*Insert ➤ Insert Cell Above or Insert ➤ Insert Cell Below*
✂	Cutting a selected cell	*Esc+x*	Edit ➤ Cut Cells
⧉	Copying the selected cell	*Esc+c*	Edit ➤ Copy Cells
📋	Pasting a cell above or below another selected cell	*Esc+v*	*Edit ➤ Paste Cells Above* or *Edit ➤ Paste Cells Below*
⏭ Run	Running a given cell	*Ctrl+Enter* (to run selected cell); *Shift+Enter* (to run selected cell and insert a new cell)	*Cell ➤ Run Selected Cells*
■	Interrupting the kernel	*Esc+ii*	*Kernel ➤ Interrupt*
↻	Rebooting the kernel	*Esc+00*	*Kernel ➤ Restart*

If you are not sure about which keyboard shortcut to use, go to: *Help ➤ Keyboard Shortcuts*, as shown in Figure 1-8.

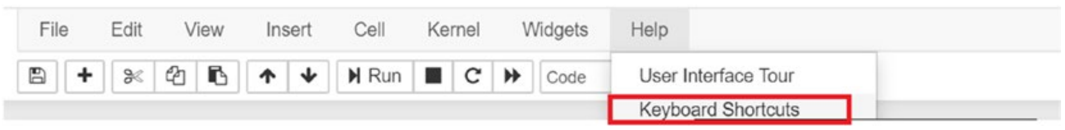


***Figure 1-8.** Help menu in Jupyter*

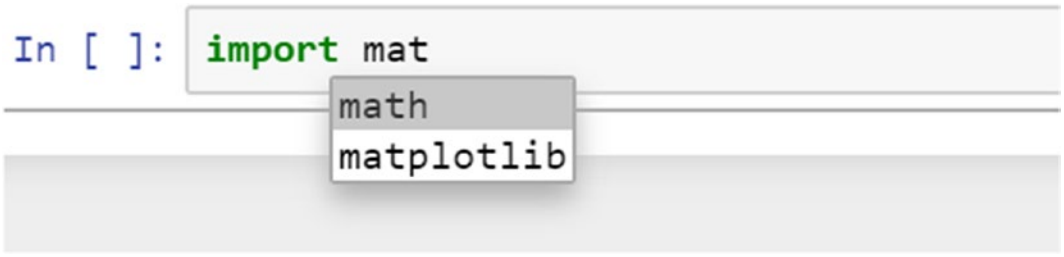
Commonly used keyboard shortcuts include

- *Shift+Enter* to run the code in the current cell and move to the next cell.
- *Esc* to leave a cell.
- *Esc+M* changes the mode for a cell to "Markdown" mode.
- *Esc+Y* changes the mode for a cell to "Code".

Tab Completion

This is a feature that can be used in Jupyter notebooks to help you complete the code being written. Usage of tab completions can speed up the workflow, reduce bugs, and quickly complete function names, thus reducing typos and saving you from having to remember the names of all the modules and functions.

For example, if you want to import the Matplotlib library but don't remember the spelling, you could type the first three letters, mat, and press Tab. You would see a drop-down list, as shown in Figure 1-9. The correct name of the library is the second name in the drop-down list.



```
In [ ]: import mat
```

Figure 1-9. *Tab completion in Jupyter*

Magic commands used in Jupyter

Magic commands are special commands that start with one or more % signs, followed by a command. The commands that start with one % symbol are applicable for a single line of code, and those beginning with two % signs are applicable for the entire cell (all lines of code within a cell).

One commonly used magic command, shown in the following, is used to display Matplotlib graphs inside the notebook. Adding this magic command avoids the need to call the *plt.show* function separately for showing graphs (the Matplotlib library is discussed in detail in Chapter 7).

CODE:

```
%matplotlib inline
```

Magic commands, like *timeit*, can also be used to time the execution of a script, as shown in the following.

CODE:

```
%%timeit
for i in range(100000):
    i*i
```

Output:

```
16.1 ms ± 283 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Now that you understand the basics of using Jupyter notebooks, let us get started with Python and understand the core aspects of this language.

Python Basics

In this section, we get familiar with the syntax of Python, commenting, conditional statements, loops, and functions.

Comments, print, and input

In this section, we cover some basics like printing, obtaining input from the user, and adding comments to help others understand your code.

Comments

A comment explains what a line of code does, and is used by programmers to help others understand the code they have written. In Python, a comment starts with the # symbol.

Proper spacing and indentation are critical in Python. While other languages like Java and C++ use brackets to enclose blocks of code, Python uses an indent of four spaces to specify code blocks. One needs to take care of indents to avoid errors. Applications like Jupyter generally take care of indentation and automatically add four spaces at the beginning of a block of code.

Printing

The *print* function prints content to the screen or any other output device.

Generally, we pass a combination of strings and variables as arguments to the print function. Arguments are the values included within the parenthesis of a function, which the function uses for producing the result. In the following statement, “Hello!” is the argument to the *print* function.

CODE:

```
print("Hello!")
```

To print multiple lines of code, we use triple quotes at the beginning and end of the string, for example:

CODE:

```
print('''Today is a lovely day.
It will be warm and sunny.
It is ideal for hiking.''')
```

Output:

```
Today is a lovely day.
It will be warm and sunny.
It is ideal for hiking.
```

Note that we do not use semicolons in Python to end statements, unlike some other languages.

The *format* method can be used in conjunction with the *print* method for embedding variables within a string. It uses curly braces as placeholders for variables that are passed as arguments to the method.

Let us look at a simple example where we print variables using the *format* method.

CODE:

```
weight=4.5
name="Simi"
print("The weight of {} is {}".format(name,weight))
```

Output:

```
The weight of Simi is 4.5
```

The preceding statement can also be rewritten as follows without the format method:

CODE:

```
print("The weight of",name,"is","weight")
```

Note that only the string portion of the print argument is enclosed within quotes. The name of the variable does not come within quotes. Similarly, if you have any constants in your print arguments, they also do not come within quotes. In the following example, a Boolean constant (True), an integer constant (1), and strings are combined in a print statement.

CODE:

```
print("The integer equivalent of",True,"is",1)
```

Output:

```
The integer equivalent of True is 1
```

The format fields can specify precision for floating-point numbers. Floating-point numbers are numbers with decimal points, and the number of digits after the decimal point can be specified using format fields as follows.

CODE:

```
x=91.234566
print("The value of x upto 3 decimal points is {:.3f}".format(x))
```

Output:

```
The value of x upto 3 decimal points is 91.235
```

We can specify the position of the variables passed to the method. In this example, we use position “1” to refer to the second object in the argument list, and position “0” to specify the first object in the argument list.

CODE:

```
y='Jack'
x='Jill'
print("{1} and {0} went up the hill to fetch a pail of water".format(x,y))
```

Output:

```
Jack and Jill went up the hill to fetch a pail of water
```

Input

The *input* function accepts inputs from the user. The input provided by the user is stored as a variable of type *String*. If you want to do any mathematical calculations with any numeric input, you need to change the data type of the input to int or float, as follows.

CODE:

```
age=input("Enter your age:")
print("In 2010, you were",int(age)-10,"years old")
```

Output:

```
Enter your age:76
In 2010, you were 66 years old
```

Further reading on Input/Output in Python: https://docs.python.org/3/tutorial/inputoutput.html

Variables and Constants

A constant or a literal is a value that does not change, while a variable contains a value can be changed. We do not have to declare a variable in Python, that is, specify its data type, unlike other languages like Java and C/C++. We define it by giving the variable a name and assigning it a value. Based on the value, a data type is automatically assigned to it. Values are stored in variables using the assignment operator (=). The rules for naming a variable in Python are as follows:

- a variable name cannot have spaces
- a variable cannot start with a number

- a variable name can contain only letters, numbers, and underscore signs (_)
- a variable cannot take the name of a reserved keyword (for example, words like *class*, *continue*, *break*, *print*, etc., which are predefined terms in the Python language, have special meanings, and are invalid as variable names)

Operators

The following are some commonly used operators in Python.

Arithmetic operators: Take two integer or float values, perform an operation, and return a value.

The following arithmetic operators are supported in Python:

- **(Exponent)
- %(modulo or remainder),
- //(quotient),
- *(multiplication)
- -(subtraction)
- +(addition)

The order of operations is essential. Parenthesis takes precedence over exponents, which takes precedence over division and multiplication, which takes precedence over addition and subtraction. An acronym was designed - P.E.D.M.A.S.(Please Excuse My Dear Aunt Sally) - that can be used to remember the order of these operations to understand which operator first needs to be applied in an arithmetic expression. An example is given in the following:

CODE:

```
(1+9)/2-3
```

Output:

```
2.0
```

In the preceding expression, the operation inside the parenthesis is performed first, which gives 10, followed by division, which gives 5, and then subtraction, which gives the final output as 2.

Comparison operators: These operators compare two values and evaluate to a true or false value. The following comparison operators are supported in Python:

- >: Greater than
- < : Less than
- <=: Less than or equal to
- >=: Greater than or equal to
- == : equality. Please note that this is different from the assignment operator (=)
- !=(not equal to)

Logical (or Boolean) operators: Are similar to comparison operators in that they also evaluate to a *true* or *false* value. These operators operate on Boolean variables or expressions. The following logical operators are supported in Python:

- *and operator*: An expression in which this operator is used evaluates to *True* only if all its subexpressions are *True*. Otherwise, if any of them is *False*, the expression evaluates to *False*
  An example of the usage of the *and* operator is shown in the following.
  CODE:

  ```
  (2>1) and (1>3)
  ```

  Output:

  ```
  False
  ```

- *or* operator: An expression in which the *or* operator is used, evaluates to *True* if any one of the subexpressions within the expression is *True*. The expression evaluates to *False* if all its subexpressions evaluate to *False.*

  An example of the usage of the *or* operator is shown in the following.

  CODE:

  ```
  (2>1) or (1>3)
  ```

Output:

```
True
```

- *not* operator: An expression in which the *not* operator is used, evaluates to *True* if the expression is *False*, and vice versa.

  An example of the usage of the *not* operator is shown in the following.
  CODE:

  ```
  not(1>2)
  ```

  Output:

  ```
  True
  ```

Assignment operators

These operators assign a value to a variable or an operand. The following is the list of assignment operators used in Python:

- = (assigns a value to a variable)
- += (adds the value on the right to the operand on the left)
- -= (subtracts the value on the right from the operand on the left)
- *= (multiplies the operand on the left by the value on the right)
- %= (returns the remainder after dividing the operand on the left by the value on the right)
- /= (returns the quotient, after dividing the operand on the left by the value on the right)
- //= (returns only the integer part of the quotient after dividing the operand on the left by the value on the right)

Some examples of the usage of these assignment operators are given in the following.

CODE:

```
x=5 #assigns the value 5 to the variable x
x+=1 #statement adds 1 to x (is equivalent to x=x+1)
x-=1 #statement subtracts 1 from x (is equivalent to x=x-1)
x*=2 #multiplies x by 2(is equivalent to x=x*2)
```

```
x%=3 #equivalent to x=x%3, returns remainder
x/=3 #equivalent to x=x/3, returns both integer and decimal part of quotient
x//=3 #equivalent to x=x//3, returns only the integer part of quotient
after dividing x by 3
```

Identity operators (is and not is)

These operators check for the equality of two objects, that is, whether the two objects point to the same value and return a Boolean value (*True/False*) depending on whether they are equal or not. In the following example, the three variables *"x"*, *"y"*, and *"z"* contain the same value, and hence, the identity operator (*is*) returns *True* when "x" and "z" are compared.

Example:

```
x=3
y=x
z=y
x is z
```

Output:

```
True
```

Membership operators (in and not in)

These operators check if a particular value is present in a string or a container (like lists and tuples, discussed in the next chapter). The *in* operator returns "True" if the value is present, and the *not in* operator returns "True" if the value is not present in the string or container.

CODE:

```
'a' in 'and'
```

Output:

```
True
```

Data types

The data type is the category or the type of a variable, based on the value it stores.

The data type of a variable or constant can be obtained using the *type* function.

CODE:

```
type(45.33)
```

Output:

```
float
```

Some commonly used data types are given in Table 1-2.

Table 1-2. *Common Data Types in Python*

Type of data	Data type	Examples

Numeric data	*int*: for numbers without a decimal point
	float: for numbers with a decimal point
		`#int`
		`a=1`
		`#float`
		`b=2.4`

Sequences	Sequences store more than one value. Some of the sequences in Python are:
	• *list*
	• *range*
	• *tuple*
		`#tuple`
		`a=(1,2,3)`
		`#list`
		`b=[1,2,3]`
		`#range`
		`c=range(5)`

Characters or text	*str* is the data type for storing a single character or a sequence of characters within quotes	`#single character`
		`X='a'`
		`#multiple characters`
		`x='hello world'`
		`#multiple lines`
		`x='''hello world`
		`good morning'''`

Boolean data	*bool* is the data type for storing True or False values	`X=True`
		`Y=False`

Mapping objects	*dict* is the data type for a dictionary (an object mapping a key to a value)	`x={'Apple':'fruit',`
		`'Carrot':'vegetable'}`

Representing dates and times

Python has a module called *datetime* that allows us to define a date, time, or duration.

We first need to import this module so that we can use the functions available in this module for defining a date or time object, using the following statement.

CODE:

```
import datetime
```

Let us use the methods that are part of this module to define various date/time objects.

Date object

A date consisting of a day, month, and year can be defined using the *date* method, as shown in the following.

CODE:

```
date=datetime.date(year=1995,month=1,day=1)
print(date)
```

Output:

```
1995-01-01
```

Note that all three arguments of the *date* method – day, month, and year – are mandatory. If you skip any of these arguments while defining a *date* object, an error occurs, as shown in the following.

CODE:

```
date=datetime.date(month=1,day=1)
print(date)
```

Output:

```
TypeError                         Traceback (most recent call last)
<ipython-input-3-7da76b18c6db> in <module>
----> 1 date=datetime.date(month=1,day=1)
      2 print(date)

TypeError: function missing required argument 'year' (pos 1)
```

Time object

To define an object in Python that stores time, we use the *time* method.

The arguments that can be passed to this method may include hours, minutes, seconds, or microseconds. Note that unlike the *date* method, arguments are not mandatory for the *time* method (they can be skipped).

CODE:

```
time=datetime.time(hour=12,minute=0,second=0,microsecond=0)
print("midnight:",time)
```

Output:

```
midnight: 00:00:00
```

Datetime object

We can also define a datetime object consisting of both a date and a time, using the *datetime* method, as follows. For this method, the date arguments – day, month, and year – are mandatory, but the time argument (like hour, minute, etc.) can be skipped.

CODE:

```
datetime1=datetime.datetime(year=1995,month=1,day=1,hour=12,minute=0,second
=0,microsecond=0)
print("1st January 1995 midnight:", datetime1)
```

Output:

```
1st January 1995 midnight: 1995-01-01 12:00:00
```

Timedelta object

A *timedelta* object represents a specific duration of time, and is created using the *timedelta* method.

Let us create a *timedelta* object that stores a period of 17 days.

CODE:

```
timedelta1=datetime.timedelta(weeks=2,days=3)
timedelta1
```

Output:

```
datetime.timedelta(days=17)
```

You can also add other arguments like seconds, minutes, and hours, while creating a *timedelta* object.

A *timedelta* object can be added to an existing date or datetime object, but not to a time object

Adding a duration (*timedelta* object) to a *date* object:

CODE:

```
#adding a duration to a date object is supported
date1=datetime.date(year=1995,month=1,day=1)
timedelta1=datetime.timedelta(weeks=2,days=3)
date1+timedelta1
```

Output:

```
datetime.date(1995, 1, 18)
```

Adding a duration (*timedelta* object) to a *datetime* object:

CODE:

```
#adding a duration to a datetime object is supported
datetime1=datetime.datetime(year=1995,month=2,day=3)
timedelta1=datetime.timedelta(weeks=2,days=3)
datetime1+timedelta1
```

Output:

```
datetime.datetime(1995, 2, 20, 0, 0)
```

Adding a duration to a *time* object leads to an error:

CODE:

```
#adding a duration to a time object is not supported
time1=datetime.time(hour=12,minute=0,second=0,microsecond=0)
timedelta1=datetime.timedelta(weeks=2,days=3)
time1+timedelta1
```

Output:

```
TypeError                                 Traceback (most recent call last)
<ipython-input-9-5aa64059a69a> in <module>
      2 time1=datetime.time(hour=12,minute=0,second=0,microsecond=0)
      3 timedelta1=datetime.timedelta(weeks=2,days=3)
----> 4 time1+timedelta1

TypeError: unsupported operand type(s) for +: 'datetime.time' and
'datetime.timedelta'
```

Further reading:
Learn more about the Python datetime module
https://docs.python.org/3/library/datetime.html

Working with Strings

A string is a sequence of one or more characters enclosed within quotes (both single and double quotes are acceptable). The data type for strings is *str.* Python does not support the character data type, unlike older languages like Java and C. Even single characters, like 'a', 'b', are stored as strings. Strings are internally stored as arrays and are immutable (cannot be modified). Let us see how to define a string.

Defining a string

Single-line strings can be defined using single or double quotes.

CODE:

```
x='All that glitters is not gold'
#OR
x="All that glitters is not gold"
```

For multiline strings, use triple quotes:

CODE:

```
x='''Today is Tuesday.
Tomorrow is Wednesday'''
```

String operations

Various functions can be used with strings, some of which are explained in the following.

1. Finding the length of a string: The *len* function can be used to calculate the length of a string, as shown in the following.

   CODE:

   ```
   len('Hello')
   ```

   Output:

   ```
   5
   ```

2. Accessing individual elements in a string:

   The individual characters in a string can be extracted using the indexing operator, [].

   CODE:

   ```
   x='Python'
   x[3]
   Output:
   'h'
   ```

3. Slicing a string: Slicing refers to the extraction of a portion or subset of an object (in this case, the object is a string). Slicing can also be used with other iterable objects like lists and tuples, which we discuss in the next chapter. The colon operator is used for slicing, with an optional start, stop, and step index. Some examples of slicing are provided in the following.

   CODE:

   ```
   x='Python'
   ```

   ```
   x[1:] #from second character to the end
   ```

   Output:

   ```
   'ython'
   ```

Some more examples of slicing:

CODE:

```
x[:2] #first two characters. The starting index is assumed to be 0
```

Output:

```
'Py'
```

CODE:

```
x[::-1]#reversing the string, the last character has an index -1
```

Output:

```
'nohtyP'
```

4. Justification:

   To add spaces to the right or left, or center the string, the *rjust*, *ljust*, or *center* method is used. The first argument passed to such a method is the length of the new string, and the optional second argument is the character to be used for padding. By default, spaces are used for padding.

   CODE:

   ```
   '123'.rjust(5,"*")
   ```

   Output:

   ```
   '**123'
   ```

5. Changing the case: To change the case of the string, the *upper* or *lower* method is used, as shown in the following.
   CODE:

   ```
   'COLOR'.lower()
   ```

   Output:

   ```
   'color'
   ```

6. Checking what a string contains:

   In order to check whether a string starts or ends with a given character, the *startswith* or *endswith* method is used.
   CODE:

   ```
   'weather'.startswith('w')
   ```

   Output:

   ```
   True
   ```

7. Removing whitespaces from a string:

   To remove spaces from a string, use the *strip* method (to remove spaces at both ends), *rstrip* (to remove spaces from the right end), or the *lstrip* method (to remove spaces from the left end). An example is shown in the following.

   CODE:

   ```
   '  Hello'.lstrip()
   ```

   Output:

   ```
   'Hello'
   ```

8. Examining the contents of a string:

   There are several methods to check what a string contains, like `isalpha, isupper, isdigit, isalnum`, etc. All these methods return "True" only if all the characters in the string satisfy a given condition.

   CODE:

   ```
   '981'.isdigit()#to check for digits
   ```

   Output:

   ```
   True
   ```

CODE:

```
'Abc'.isupper()
#Checks if all characters are in uppercase. Since all letters are
not uppercase, the condition is not satisfied
```

Output:

```
False
```

9. Joining a list of strings:

   The *join* method combines a list of strings into one string. On the left-hand side of the *join* method, we mention the delimiter in quotes to be used for joining the strings. On the right-hand side, we pass the list of individual strings.

   CODE:

   ```
   ' '.join(['Python','is','easy','to','learn'])
   ```

   Output:

   ```
   'Python is easy to learn'
   ```

10. Splitting a string:

    The *split* method does the opposite of what the join method does. It breaks down a string into a list of individual words and returns the list. If we just pass one word to this method, it returns a list containing just one word and does not split the string further.

    CODE:

    ```
    'Python is easy to learn'.split()
    ```

    Output:

    ```
    ['Python', 'is', 'easy', 'to', 'learn']
    ```

Conditional statements

Conditional statements, as the name indicates, evaluate a condition or a group of conditions. In Python, the *if-elif-else* construct is used for this purpose. Python does not have the switch-case construct, which is used in some other languages for conditional execution.

Conditional statements start with the *if* keyword, and the expression or a condition to be evaluated. This is followed by a block of code that executes only if the condition evaluates to "True"; otherwise it is skipped.

The *else* statement (which does not contain any condition) is used to execute a block of code when the condition mentioned in the *if* statement is not satisfied. The *elif* statements are used to evaluate specific conditions. The order of *elif* statements matters. If one of the *elif* statements evaluates to True, the *elif* statements following it are not executed at all. The *if* statement can also exist on its own, without mentioning the *else* or *elif* statements.

The following example demonstrates the *if-elif-else* construct.

CODE:

```
#if-elif-else
color=input('Enter one of the following colors - red, orange or blue:')
if color=='red':
    print('Your favorite color is red')
elif color=='orange':
    print('Your favorite color is orange')
elif color=='blue':
    print('Your favorite color is blue')
else:
    print("You entered the wrong color")
```

Output:

```
Enter one of the following colors - red, orange or blue:pink
You entered the wrong color
```

Conditional statements can be nested, which means that we can have one conditional statement (inner) within another (outer). You need to be particularly careful with indentation while using nested statements. An example of nested *if* statements is shown in the following.

CODE:

```
#nested conditionals
x=20
if x<10:
    if x<5:
        print("Number less than 5")
    else:
        print("Number greater than 5")
else:
    print("Number greater than 10")
```

Output:

```
Number greater than 10
```

Further reading: See more on the *if* statement: https://docs.python.org/3/tutorial/controlflow.html#if-statements

Loops

Loops are used to execute a portion of the code repeatedly. A single execution of a block of code is called an iteration, and loops often go through multiple rounds of iterations. There are two types of loops that are used in Python – the *for* loop and the *while* loop.

While loop

The *while* loop is used when we want to execute particular instructions as long as a condition is "True". After the block of code executes, the execution goes back to the beginning of the block. An example is shown in the following.

CODE:

```
#while loop with continue statement
while True:
    x=input('Enter the correct color:')
    if(x!='red'):
        print("Color needs to be entered as red")
        continue
    else:
        break
```

Output:

```
Enter the correct color:blue
Color needs to be entered as red
Enter the correct color:yellow
Color needs to be entered as red
Enter the correct color:red
```

In the preceding example, the first statement (*while True*) is used to execute an infinite loop. Once the username entered is of the right length, the break statement takes execution outside the loop; otherwise, a message is displayed to the user asking for a username of the right length. Note that execution automatically goes to the beginning of the loop, after the last statement in the block of code.

The *break* statement is used to take the control outside the loop. It is useful when we have an infinite loop that we want to break out of.

The *continue* statement does the opposite - it takes control to the beginning of the loop. The keywords *break* and *continue* can be used both with loops and conditional statements, like *if/else*.

Further reading:

See more about the following:

- *break* and *continue* statements: `https://docs.python.org/3/tutorial/controlflow.html#break-and-continue-statements-and-else-clauses-on-loops`
- *while* statement: `https://docs.python.org/3/reference/compound_stmts.html#while`

for loop

The *for* loop is used to execute a block of a code a predetermined number of times. The *for* loop can be used with any kind of iterable object, that is, a sequence of values that can be used by a loop for running repeated instances or iterations. These iterable objects include lists, tuples, dictionaries, and strings.

The *for* loop is also used commonly in conjunction with the *range* function. The range function creates a *range* object, another iterable, which is a sequence of evenly spaced integers. Consider the following example where we calculate the sum of the first five odd integers using a *for* loop.

CODE:

```
#for loop
sum=0
for i in range(1,10,2):
    sum=sum+i
print(sum)
```

Output:

```
25
```

The *range* function has three arguments: the start argument, the stop argument, and the step argument. None of these three arguments are mandatory. Numbers from 0 to 9 (both 0 and 9 included) can be generated as *range(10)*, *range(0,10)*, or *range(0,10,1)*. The default start argument is 0, and the default step argument is 1.

For loops can also be nested (with an outer loop and any number of inner loops), as shown in the following.

CODE:

```
#nested for loop
for i in 'abcd':
    for j in range(4):
        print(i,end=" ")
    print("\n")
```

Output:

```
a a a a

b b b b

c c c c

d d d d
```

Further reading: See more about the for statement: https://docs.python.org/3/tutorial/controlflow.html#for-statements

Functions

A function can be thought of as a "black box" (the user need not be concerned with the internal workings of the function) that takes an input, processes it, and produces an output. A function is essentially a block of statements performing a specific task.

In Python, a function is defined using the *def* keyword. This is followed by the name of a function and one or more optional parameters. A parameter is a variable that exists only within a function. Variables defined within a function have local scope, which means that they cannot be accessed outside the function. They are also called local variables. External code or functions cannot manipulate the variables defined within a function.

A function may have an optional return value. The return value is the output produced by a function that is returned to the main program. Calling a function means giving the function inputs (arguments) to perform its task and produce an output.

The utility of functions lies in their reusability. They also help in avoiding redundancy and organizing code into logical blocks. We just need to supply it with the set of inputs it needs to run the instructions. A function can be called repeatedly instead of manually typing out the same lines of code.

For example, say you want to find out the prime numbers in a given list of numbers. Once you have written a function for checking whether an integer is a prime number, you can simply pass each number in the list as an argument to the function and call it, instead of writing the same lines of code for each integer you want to test.

CODE:

```
def checkPrime(i):
     #Assume the number is prime initially
    isPrime=True
    for j in range(2,i):
        # checking if the number is divisible by any number between 2 and i
        if i%j==0:
            #If it is divisible by any number in the j range, it is not prime
            isPrime=False
    # This is the same as writing if isPrime==True
    if isPrime:
        print(i ,"is prime")
    else:
        print(i, "is not prime")
for i in range(10,20):
    checkPrime(i)
```

Output:

```
10 is not prime
11 is prime
12 is not prime
13 is prime
14 is not prime
15 is not prime
16 is not prime
17 is prime
18 is not prime
19 is prime
```

Further reading: See more about defining functions: https://docs.python.org/3/tutorial/controlflow.html#defining-functions

Anonymous or lambda functions are defined using the *lambda* keyword. They are single-expression functions and provide a compact way of defining a function without binding the function object to a name. The reason these functions are called "anonymous" is that they do not need a name. Consider the following example where we use a lambda function to calculate the sum of two numbers.

CODE:

```
(lambda x,y:(x+y))(5,4)
```

Output:

```
9
```

Note the syntax of an anonymous function. It starts with the *lambda* keyword, followed by the parameters ('x' and 'y', in this case). Then comes the colon, after which there is an expression that is evaluated and returned. There is no need to mention a return statement since there is an implicit return in such a function. Notice that the function also does not have a name.

Syntax errors and exceptions

Syntax errors are errors that may be committed inadvertently by the user while writing the code, for example, spelling a keyword wrong, not indenting the code, and so on. An exception, on the other hand, is an error that occurs during program execution. A user may enter incorrect data while running the program. If you want to divide a number (say, 'a') by another number (say, 'b'), but give a value of 0 to the denominator ('b'), this will generate an exception. The exceptions, which are autogenerated in Python and displayed to the user, may not lucidly convey the problem. Using exception handling with the *try-except* construct, we can frame a user-friendly message to enable the user to better correct the error.

There are two parts to exception handling. First, we put the code that is likely to cause an error under a *try* clause. Then, in the *except* clause, we try to deal with whatever caused an error in the *try* block. We mention the name of the exception class in the *except* clause, followed by a code block where we handle the error. A straightforward method for handling the error is printing a message that gives the user more details on what they need to correct.

Note that all exceptions are objects that are derived from the class *BaseException*, and follow a hierarchy.

Further reading: The class hierarchy for exceptions in Python can be found here: https://docs.python.org/3/library/exceptions.html#exception-hierarchy

A simple example of a program, with and without exception handling, is shown below.

```
while True:
    try:
        n=int(input('Enter your score:'))
        print('You obtained a score of ',n)
        break
    except ValueError:
        print('Enter only an integer value')
```

Output:

```
Enter your score(without a decimal point):abc
Enter only an integer value
Enter your score(without a decimal point):45.5
Enter only an integer value
Enter your score(without a decimal point):90
You obtained a score of  90
```

Same program (Without exception handling):

CODE:

```
n=int(input('Enter your score:'))
print('You obtained a score of ',n)
```

Output:

```
Enter your score:ninety three
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-12-aa4fbda9d45f> in <module>
----> 1 n=int(input('Enter your score:'))
      2 print('You obtained a score of ',n)

ValueError: invalid literal for int() with base 10: 'ninety three'
```

The statement that is likely to cause an error in the preceding code is: `int(input('Enter your score:'))`. The *int* function requires an integer as an argument. If the user enters a floating-point or string value, a *ValueError* exception is generated. When we use the *try-except* construct, the *except* clause prints a message asking the user to correct the input, making it much more explicit.

Working with files

We can use methods or functions in Python to read or write to files. In other words, we can create a file, add content or text to it, and read its contents by using the methods provided by Python.

Here, we discuss how to read and write to comma-separated value (CSV) files. CSV files or comma-separated files are text files that are a text version of an Excel spreadsheet.

The functions for all of these operations are defined under the CSV module. This module has to be imported first, using the `import csv` statement, to use its various methods.

Reading from a file

Reading from a file from Python involves the following steps:

1. Using the `with open` statement, we can open an existing CSV file and assign the resulting file object to a variable or file handle (named 'f' in the following example). Note that we need to specify the path of the file using either the absolute or relative path. After this, we need to specify the mode for opening the file. For reading, the mode is 'r'. The file is opened for reading by default if we do not specify a mode.
2. Following this, there is a block of code that starts with storing the contents of the file in a read object, using the *csv.reader* function where we specify the file handle, f, as an argument.
3. However, the contents of this file are not directly accessible through this read object. We create an empty list (named 'contents' in the following example), and then we loop through the read object we created in step 2 line by line using a for loop and append it to this list. This list can then be printed to view the lines of the CSV file we created.

CODE:

```
#Reading from a file
import csv
with open('animals.csv') as f:
    contents=csv.reader(f)
    lines_of_file=[]
    for line in contents:
        lines_of_file+=line
lines_of_file
```

Writing to a file

Writing to a file involves the following steps.

1. Using the *open* function, open an existing CSV file or if the file does not exist, the open function creates a new file. Pass the name of the file (with the absolute path) in quotes and specify the mode as 'w', if you want to overwrite the contents or write into a new file. Use the 'a' or 'append' mode if you simply want to append some lines to an existing file. Since we do not want to overwrite in this case, we open the file using the append ('a') mode. Store it in a variable or file handle and give it a name, let us say 'f'.
2. Using the *csv.writer()* function, create a writer object to add the content since we cannot directly write to the CSV file. Pass the variable (file handle), 'f', as an argument to this function.
3. Invoke the *writerow* method on the writer object created in the previous step. The argument to be passed to this method is the new line to be added (as a list).
4. Open the CSV file on your system to see if the changes have been reflected.

CODE:

```
#Writing to a file
with open(r'animals.csv',"w") as f:
    writer_object=csv.writer(f,delimiter=",")
    writer_object.writerow(['sheep','lamb'])
```

The modes that can be used with the open function to open a file are:

- "r": opens a file for only reading.
- "w": opens a file for only writing. It overwrites the file if it already exists.
- "a": opens a file for writing at the end of the file. It retains the original file contents.
- "w+": opens the file for both reading and writing.

Further reading: See more about reading and writing to files in Python: https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files

Modules in Python

A module is a Python file with a .py extension. It can be thought of as a section of a physical library. Just as each section of a library (for instance, fiction, sports, fitness) contains books of a similar nature, a module contains functions that are related to one another. For example, the *matplotlib* module contains all functions related to plotting graphs. A module can also contain another module. The *matplotlib* module, for instance, contains a module called *pyplot*. There are many built-in functions in Python that are part of the standard library and do not require any module to be imported to use them.

A module can be imported using the *import* keyword, followed by the name of the module:

CODE:

```
import matplotlib
```

You can also import part of a module (a submodule or a function) using the *from* keyword. Here, we are importing the cosine function from the math module:

CODE:

```
from math import cos
```

Creating and importing a customized module in Python requires the following steps:

1. Type "idle" in the search bar next to the start menu. Once the Python shell is open, create a new file by selecting: *File* ➤ *New File*
2. Create some functions with similar functionality. As an example, here, we are creating a simple module that creates two functions - sin_angle and cos_angle. These functions calculate the sin and cosine of an angle (given in degrees).

   CODE:

   ```
   import math
   def sin_angle(x):
       y=math.radians(x)
       return math.sin(y)
   ```

```
def cos_angle(x):
    y=math.radians(x)
    return math.cos(y)
```

3. Save the file. This directory, where the file should be saved, is the same directory where Python runs. You can obtain the current working directory using the following code:

   CODE:

   ```
   import os
   os.getcwd()
   ```

4. Using the import statement, import and use the module you have just created.

Python Enhancement Proposal (PEP) 8 – standards for writing code

Python Enhancement Proposal (PEP) is a technical document that provides documentation for new features introduced in the Python language. There are many types of PEP documents, the most important among these being PEP 8. The PEP 8 document provides style guidelines for writing code in Python. The main emphasis of PEP 8 is on providing a set of consistent rules that enhance code readability – anybody who reads your code should be able to understand what you are doing. You can find the complete PEP8 document here: https://www.python.org/dev/peps/pep-0008/

There are several guidelines in PEP8 for different aspects of the code, some of which are summarized in the following.

- Indentation: Indentation is used to indicate the starting of a block of code. In Python, four spaces are used for indentation. Tabs should be avoided for indentation.
- Line length: The maximum character length for a line of code is 79 characters. For comments, the limit is 72 characters.

- The naming conventions for naming different types of objects in Python are also laid out in PEP 8. Short names should be used, and underscores can be used to improve readability. For naming functions, methods, variables, modules, and packages, use the lowercase (all small letters) notation. With constants, use the uppercase (all capitals) notation, and for classes, use the CapWords (two words each starting with a capital letter, not separated by spaces) notation for naming.
- Comments: Block comments, starting with a # and describing an entire block of code, are recommended. Inline comments, which are on the same line as the line of code, as shown in the following, should be avoided. If they are used at all, they should be separated by two spaces from the code.

  CODE:

  ```
  sum+=1 #incrementing the sum by 1
  ```

- Imports:

  While importing a module to use it in your code, avoid wildcard imports (using the * notation), like the one shown in the following.

  CODE:

  ```
  from module import *
  ```

  Multiple packages or classes should not be imported on the same line.

  CODE:

  ```
  import a,b
  ```

  They should be imported on separate lines, as shown in the following.

  CODE:

  ```
  import a
  import b
  ```

Absolute imports should be used as far as possible, for example:

CODE:

```
import x.y
```

Alternatively, we can use this notation for importing modules:

CODE:

```
from x import y
```

- Encoding: The encoding format to be used for writing code in Python 3 is UTF-8

Summary

- The syntax of Python differs from other languages like Java and C, in that statements in Python do not end with a semicolon, and spaces (four spaces), are used for indentation, instead of curly braces.
- Python has basic data types like *int*, *float*, *str*, and *bool*, among many others, and operators (arithmetic, Boolean, assignment, and comparison) that operate on variables depending on their data type.
- Python has the *if-elif-else* keywords for the conditional execution of statements. It also has the *for* loop and the *while* loop for repeating a specific portion of the program.
- Functions help with reusing a part of code and avoiding redundancy. Each function should perform only one task. Functions in Python are defined using the *def* keyword. Anonymous or *lambda* functions provide a shortcut for writing functions in a single line without binding the function to a name.
- A module is a collection of similar functions and is a simple Python file. Apart from the functions that are part of the standard library, any function that is part of an external module requires the module to be imported using the *import* keyword.

- Python has functions for creating, reading, and writing to text and CSV files. The files can be opened in various modes, depending on whether you want to read, write, or append data.
- Exception handling can be used to handle exceptions that occur during the execution of the program. Using the *try* and *except* keywords, we can deal with the part of the program that is likely to cause an exception.
- PEP 8 sets standards for a range of coding-related aspects in Python, including usage of spaces, tabs, and blank lines, and conventions for naming and writing comments.

The next chapter delves deep into topics like containers, like lists, tuples, dictionaries, and sets. We also discuss a programming paradigm known as object-oriented programming, and how it is implemented using classes and objects.

Review Exercises

Question 1

Calculate the factorial of numbers from 1 to 5 using nested loops.

Question 2

A function is defined using which of the following?

1. *def* keyword
2. *function* keyword
3. *void* keyword
4. No keyword is required

Question 3

What is the output of the following code?

```
x=True
y=False
z=True
x+y+z
```

Question 4

Write a Python program to print the following sequence:

```
*****

****

***

**

*
```

Question 5

Which of these variables has been defined using the correct syntax?

1. `1x=3`
2. `x 3=5`
3. `x`
4. `x_3=5`
5. `x$4=5`

Question 6

What is the output of the following code? (Hint: The id function returns the memory address of an object.)

```
str1="Hello"
str2=str1
id(str1)==id(str2)
```

Question 7

Convert the string “123-456-7890” into the format “1234567890”. Use the *join* and *split* string functions.

Question 8

Write a function that performs the following tasks (the name of the file is passed as a parameter):

- Create a new text file with the name passed as an argument to the function
- Add a line of text ("Hello World") to the file
- Read the contents of the file
- Opens the file again, add another line ("This is the next line") below the first line
- Reread the file and print the contents on the screen

Answers

Question 1

Solution:

```
#Question 1
for i in range(1,6):
    fact=1
    for j in range(1,i+1):
        fact=fact*j
    print("Factorial of number ",i," is:",fact)
```

Question 2

Option 1: Functions in Python require the *def* keyword.

Question 3

Output: 2

Explanation: The Boolean value "True" is treated as value 1, and 'False' as value 0. Applying the addition operator on Boolean variables is valid.

Question 4

Solution

```
#question 4
l=range(6)
for i in l[::-1]:
    print("*"*i)
    print("\n")
```

Question 5

Option 4 is correct.

Let us go through the options, one by one:

1. 1x=3: incorrect, as a variable cannot start with a number
2. x 3=5: incorrect, as a variable name cannot contain a space
3. x : incorrect, as a variable needs an initial value
4. x_3=5: correct; underscore is an acceptable character while defining variables
5. x$4=5: incorrect; special characters like $ are not permissible

Question 6

Both the strings have the same value and memory address.

Output:

```
True
```

Question 7

This problem can be solved in one line – simply split the string, convert it to a list, and join it back into a string.

CODE:

```
"".join(list("123-456-7890".split("-")))
```

Question 8

Solution:

```
#Question 8
def filefunction(name):
    #open the file for writing
    with open(name+".txt","w") as f:
        f.write("Hello World")
    #read and print the file contents
    with open(name+".txt","r") as f:
        print(f.read())
    #open the file again the append mode
    with open(name+".txt","a") as f:
        f.write("\nThis is the next line")
    #reread and print the lines in the file
    with open(name+".txt","r") as f:
        print(f.read())
filename=input("Enter the name of the file ")
filefunction(filename)
```